THEORETICAL CONSIDERATIONS IN TESTING PROGRAMS BY DEMONSTRATING CONSISTENCY WITH SPECIFICATIONS

Debra J. Richardson

COINS Technical Report 78-21
December 1978

Department of Computer and Information Science
University of Massachusetts, Amherst
Amherst, Massachusetts 01003

This work was supported by the National Science Foundation under grant # NSFMCS 77-02101 and the U. S. Air Force under grant # AFOSR 77-3287.

Abstract

Program correctness is described in terms of the consistency between a program and a specification for the intended function of the program. Two consistency properties - equivalence and isomorphism - are defined which differ in the degree of conformity of the program to its functional specification. Given a correct specification for the program's intended function, the program is correct if and only if it is equivalent to the specification. When a strict conformity to the functional specification is presupposed, isomorphism must prevail. The program testing process can, therefore, be performed by determining the consistency between a program and a correct functional specification. Methods for the comparison of certain classes of programs and functional specificaitons are proposed, and examples of their utility are given.

Introduction

The ultimate goal of program testing is the determination of whether a program is correct. In the absence of any tools to aid this process, the demonstration of program correctness is mostly intuitive, based solely on the obtainment of correct results for the program when tested for a set of input data that the tester considers representative of the program domain. This process is capable of detecting errors in a program, but not in general of establishing the absence of errors (exhaustive testing is an exception to this disclaimer). Various alternatives to the ad hoc efforts of the human tester have been proposed.

Proofs of correctness employ rigorous mathematical proofs in an attempt to verify the consistency between a program and a specification describing the intended function of the program. If a complete correct proof (including proof of termination) is acheived, the program has been shown to be correct. This process is, however, extremely complex, tedious, and expensive. Experience has shown that the proofs generated are not always accurate, and as such, are not usually worth the tremendous cost. In addition, proofs of correctness are limited to conclusions about behavior in a postulated environment, whereas testing has the advantage of providing well-founded information about a program's behavior in its actual environment.

With this in mind, several attempts have been made to aid the human tester in the selection of test data which allow one to attach a meaningful degree of reliability to a program as a result of correct test runs. In the absence of the expedient ability to verify that a program is absolutely correct, this meaningful degree of reliability is a practical goal for program testing.

This paper examines testing programs by demonstrating consistency with specifications as an attempt to attach a meaningful degree of reliability to programs which undergo such a testing procedure. A formal notation for programs, functions, and specifications is introduced and used to concisely represent program correctness and consistency with specifications. Program correctness is defined in terms of the consistency between a program and a correct specification for the intended function of the program. A program and a functional specification are equivalent if they provide the same output values for all input data. A program is isomorphic to a specification if there is a one-to-one correspondence between the two which preserves their symbolic equality.

Methods for the comparison of certain classes of programs and specifications are presented; these comparison procedures may be used in an attempt to determine the level of consistency which holds. Program errors are discussed with respect to the discrepancies between a program and a functional specification. Symbolic execution and test data generation systems are considered as methods of testing. Some possibilities of using decision tables in the testing of programs are examined. Also, an example of the determination of equivalence and isomorphism is provided. In conclusion, some areas of future research are considered.

Programs, Functions, and Symbolic Execution

In this section, a formal notation for programs and functions is introduced. This notation will later be generalized to represent specifications and will be used in the sections which follow to define program correctness and consistency between a program and specifications. In addition, the method of symbolic execution is described using this notation.

Let P be a *program* which accepts M input variables $(x_1,x_2,\ldots,x_M)$ and produces N output variables $(z_1,z_2,\ldots,z_N)$. To simplify the analysis, it is assumed throughout this paper that the input variables are distinct from the output variables. Let X denote the *domain* of the program P. The set X is a cross product, $X = X_1 \times X_2 \times \ldots \times X_M$, where each X_I is the domain for the input variable x_I. An element of X is a vector d with a specific value d_I for each of the M input variables x_I, $d = (d_1,d_2,\ldots,d_M)$, and corresponds to a single point in the M-dimensional input space X. Let Z denote the *codomain* of the program P. Likewise, the set Z is a cross product, $Z = Z_1 \times Z_2 \times \ldots \times Z_N$, where each Z_J is the codomain for the output variable z_J. Let $P(d)$ denote the values of the N output variables obtained from the execution of P on d. Then $P(d)$ is an element of Z and is a vector with specific values for each of the N output variables. $P(d)$ corresponds to a single point in the N-dimensional output space Z.

The program P computes some *function* F which has the same domain and codomain as does P; hence F is a mapping from X to Z, $F\colon\ X \to Z$. In general, F is composed of a set of *partial functions* which are defined over disjoint subsets of the domain X. Suppose $F = \{F_1, F_2,\ldots, F_T\}$, $1 \le T \le \infty$, where each F_G is a partial function defined over D^F_G and undefined elsewhere in X. This subdomain D^F_G is called the *domain of definition* of F_G. The *function domain* D^F, or domain of definition of F, is the union of all such subdomains - that is $D^F = D^F_1 \cup D^F_2 \cup\ldots\cup D^F_T$. Each of the partial functions F_G produces an N-tuple in the codomain Z, F_G may thus be represented as a vector of N *component functions*, $F_G = (f_{G1},f_{G2},\ldots,f_{GN})$, where the Jth component produces the Jth output variable, z_J. Hence, for any $d \in D^F_G$, $F(d) = F_G(d) = (f_{G1}(d),f_{G2}(d),\ldots,f_{GN}(d)) \in Z$ and $f_{GJ}(d) \in Z_J$.

A program P has a construct similar to the partial functions of the function F, which P computes. Rather than performing the same computation on all elements of the input space, P may compute different functions along different *program paths*, which are executed for disjoint subsets of the domain X. Suppose P specifies a set of program paths, $\{P_1,P_2,\ldots,P_U\}$, $1 \le U \le \infty$, where each P_H is executed for a subset D_H^P of the program domain X. This subdomain D_H^P is called the *path domain* of P_H. The *program domain* D^P, or the domain of definition of P, is the union of the path domains - that is, $D^P = D_1^P \cup D_2^P \cup \ldots \cup D_U^P$. Each program path P_H performs a computation which produces values for the N output variables. Let C_H^P denote the *path computation* which can be represented as a vector of N *path functions*, $C_H^P = (p_{H1}, p_{H2}, \ldots, p_{HN})$, where the Jth component computes the Jth output variable z_J.

Symbolic execution may be used to generate representations for the path domains and path functions of a program. In order to describe these representations and how they are generated, a few definitions must be presented.

A program can be represented by a directed graph which describes the possible flow of control through the program (this representation is typical in data flow analysis algorithms). The nodes in the graph $\{n_1, n_2, \ldots, n_w\}$ represent basic blocks (linear sequences of instructions having one entry point and one exit point), while the edges $\{(n_i, n_j), (n_k, n_l), \ldots\}$ represent possible transfers out of a given basic block. Each edge is specified by an ordered pair (n_i, n_j) of nodes, which indicates that a directed line (transfer of control) exists from node n_i to node n_j. Associated with each transfer of control are conditions under which such a transfer occurs. Let $bp[n_i, n_j]$ denote the branch predicate which governs traversal of the edge (n_i, n_j). For the purposes of this paper, the *control flow graph* of a program is such a directed graph which has a single entry point, the start node n_s, and a single exit point, the final node n_f. Both the start node and the final node are null nodes added to the graph to accomplish this single-entry, single exit form without any loss of generality.

A *path* in the control flow graph is a sequence of basic blocks, $(n_{i0}, n_{i1}, \ldots, n_{it})$, where there exists a possible transfer of control from n_{ij} to n_{ij+1} for all nodes n_{ij} in the path. A *partial program path* T_{kq} is a path which begins with the start node - that is, $T_{kq} = (n_s, n_{k1}, n_{k2}, \ldots, n_{kq})$. Hence for any partial program path T_{kq}, with $q \geq 1$, $T_{kq} = (T_{kq-1}, n_{kq})$ and $T_{k0} = (n_s)$. A *program path* P_H is a path which begins with the start node and ends with the final node - that is $P_H = (n_s, n_{H1}, n_{H2}, \ldots, n_{Hr}, n_f)$. There is no guarantee that a sequence of basic blocks representing a path is executable, some paths may be infeasible due to contradictory conditions governing the transfers of control along the path. The control flow graph is a representation of all possible paths through the corresponding program.

Symbolic execution algorithms utilize the control flow graph of a program to propogate data descriptions along program paths. A description of the program state is maintained at every point in the symbolic execution of a path. The state of the program includes a description of the path followed to reach the present point in the execution and a description of the domain of data elements which will execute this path, as well as the values obtained for all program variables following the execution of this partial program path. Given a partial program path, $T_{kq} = (n_s, n_{k1}, n_{k2}, \ldots, n_{kq})$, n_{kq} represents the present execution point. Let *VAL* $[T_{kq}]$ represent the symbolic values of all program variables after execution of the partial program path T_{kq}. *VAL*$[T_{kq}]$ is a vector containing an element for each of the M input variables x_I, each of the L intermediate variables y_K, and each of N output variables z_J. Hence, *VAL*$[T_{kq}] = (s(x_1),\ldots,s(x_M),s(y_1),\ldots,s(y_L), s(z_1),\ldots,s(z_N))$, where $s(v)$ denotes the symbolic value of the program variable v in terms of symbolic names assigned to represent the input values. Let *PC*$[T_{kq}]$ represent the path condition which is the conjunct of the evaluated branch predicates which enable control to follow this particular partial program path *PC*$[T_{kq}] = s(bp[n_s,n_{k1}](T_{k0})) \wedge s(bp[n_{k1},n_{k2}](T_{k1})) \wedge \ldots \wedge s(bp[n_{kq-1},n_{kq}](T_{kq-1}))$, alternatively, *PC*$[T_{kq}]$ = *PC*$[T_{kq-1}] \wedge s(bp[n_{kq-1},n_{kq}](T_{kq-1}))$, where $s(bp[n_{kj},n_{kj+1}](T_{kj}))$ denotes the symbolic value of the branch predicate $bp[n_{kj},n_{kj+1}]$ when evaluated over the values of the program variables preceding traversal of the corresponding edge - that is, over *VAL*$[T_{kj}]$. Finally, let *STATE*$[T_{kq}] = (T_{kq}$, *VAL*$[T_{kq}]$, *PC*$[T_{kq}])$ represent the program state following symbolic execution of the partial program path T_{kq}.

The symbolic execution of any element of the control flow graph, a basic block or a transfer of control, modifies the program state. Initially, the program state is defined as:

$T_{k0} = (n_s)$;
VAL$[T_{k0}] = (d_1,d_2,\ldots, d_M, \Lambda\ldots\Lambda)$;
PC$[T_{k0}]$ = *true*;
STATE$[T_{k0}] = (T_{k0}$, *VAL*$[T_{k0}]$ *PC*$[T_{k0}])$.

The symbolic value of an input variable, $s(x_I)$, is the symbolic name, d_I, which is assigned to represent the input value x_I. For the purposes of this paper, all input will be assumed to occur in the start node n_s (this restriction may be dropped without major modification, but the discussion is more straightforward under this assumption). All other variables have the undefined value Λ at the start node. Throughout symbolic execution, all representations of variable and branch predicate values are in terms of the initial symbolic names of the input values. This is accomplished by substituting the current symbolic value of a variable (input, intermediate, or output) into the expressions wherever that variable is referenced. Once the initial program state has been assigned, symbolic execution can proceed in a manner similar to that of normal execution.

When executing a basic block, say node n_{kj}, the program state will be updated in the *VAL* component only. The vector of symbolic values of the program variables will be modified in each component which corresponds to a variable which is assigned a new value within the basic block. For instance, if the assignment statement $z_J \leftarrow x_I * y_K$ occurs, then the z_J component of the *VAL* vector will change from its former value to the algebraic expression formed by $s(x_I) * s(y_K)$.

On the other hand, when a transfer of control, say edge (n_{kj}, n_{kj+1}), is encountered during symbolic execution, the *PC* component of the program state is updated. The branch predicate governing traversal of this edge will be evaluated and the path condition, *PC*, will be augmented by this symbolic value - that is, $PC[T_{kj+1}] = PC[T_{kj}] \wedge s(bp[n_{kj}, n_{kj+1}](T_{kj}))$.

Following symbolic execution of a complete program path, the symbolic representation of the program state defines the path functions and path domain of that particular path. Given a complete program path P_H, the program state after execution of the final node is represented as:

$$P_H = (n_s, n_{H1}, n_{H2}, \ldots, n_{Hr}, n_f);$$
$$VAL[P_H] = (s(x_1), \ldots, s(x_M), s(y_1), \ldots, s(y_L), s(z_1), \ldots, s(z_N));$$
$$PC[P_H] = s(bp[n_s, n_{H1}](T_{H0})) \wedge s(bp[n_{H1}, n_{H2}](T_{H1})) \wedge \ldots \wedge s(bp[n_{Hr}, n_f](T_{Hr}));$$
$$STATE[P_H] = (P_H, VAL[P_H], PC[P_H]).$$

The path computation and the path functions, $C_H^P = (p_{H1}, p_{H2}, \ldots, p_{HN})$, which compute the output variables $(z_1, z_2, \ldots, z_N)$ are provided by $p_{HJ} = s(z_J)$. Since all symbolic representations are in terms of the symbolic names representing the input values; p_{HJ} is a symbolic computational expression for the output variable z_J in terms of the input values $(d_1, d_2, \ldots, d_M)$. The path condition $PC[P_H]$ provides a system of constraints on the program's input variables which defines the path domain D_H^P. The subset of elements of the program domain which will cause execution of this program path is defined by $D_H^P = \{x \in X$ such that $PC[P_H]$ is true$\}$. The path computation and path domain can be generated for any program path which is symbolically executed. A method for using symbolic execution to demonstrate the consistency between a program and its specifications will be introduced shortly.

Program Correctness and Consistency with Specifications

A program is correct if it computes the intended output for all possible inputs; otherwise it contains an error.

Definition - The program P for computing the function F over domain X is correct if and only if for all $x \in X$, $P(x) = F(x)$. P contains an error if there exists $d \in X$ such that $P(d) \neq F(d)$.

In order to determine the correctness of a program, there must be some mechanism by which the intended output can be recognized. A specification describing the function that the program is to compute can be used to provide this mechanism. In addition to correct input-output relationships, a functional specification may furnish more extensive properties of the function for comparison with the program.

Program correctness can be related to the consistency between a program and a specification describing the program's intended function. There are basically two techniques for functional specifications - *input/output assertions* and *operational specifications* (11). In either case, the function is formally specified by relationships between the inputs and outputs. The technique of assertions has been used extensively in the proof of correctness approach to program verification (7). Assertions are assigned to various points in the program, including initial assertions on the input and final assertions concerning the output; rigorous mathematical proofs are employed to show that these assertions are true whenever control reaches the associated points. In program testing, the operational specifications appear to be more useful. An operational specification differs from input/output assertions because the transformation on the input variables is described explicitly by giving actions which compute the intended output of the function. Operational specifications take on several forms - program design, decision tables, input domains with associated output computations, correct programs.

Abstractly, suppose R is a specification for the function F. In order to describe the function F, the specification R must have the same vector of input variables, $x = (x_1,x_2,\ldots,x_M)$, and the same vector of output variables, $z = (z_1,z_2,\ldots,z_N)$. Given a data element $d = (d_1,d_2,\ldots,d_M)$, the specification can be applied to obtain $R(d)$, which provides the values of the output variables.

Definition - The specification R for the function F over domain X is correct if and only if for all $x \in X$, $R(x) = F(x)$.

Given a correct specification for a function, a program for computing the function can be compared with the specification for consistency. If the specification is correct and the program is consistent with the specification, then the program is correct. A program is incorrect if it is inconsistent with a correct specification.

Definition - If *R* is a correct specification for a function *F* over domain *X*, the program *P* for computing *F* over *X* is correct if and only if for all $x \in X$, $P(x) = R(x)$. *P* contains an error if there exists $d \in X$ such that $P(d) \neq R(d)$.

The correctness of a program can,therefore,be discussed in terms of its consistency with a specification that is assumed to correctly describe the intended function. In order to discuss consistency more completely, a more formal representation of a functional specification is necessary.

A functional specification *R* consists of a set of *rules* that are quite similar to the partial functions of *F* and the program paths of *P*, which were previously defined. In fact, the errors in a program will be categorized in terms of the discrepancies between the paths in a program and the rules in a functional specification for the program. A rule is actually an individual relation between a subset of the domain and the output for this set of inputs. Suppose the specification *R* designates a set of rules, $\{R_1, R_2, \ldots, R_V\}$, $1 \leq V \leq \infty$. The *rule domain* D^R_G for a particular rule R_G is the subset of the domain *X* for which the rule is applicable. The *specification domain* D^R is the union of the rule domains - that is,$D^R = D^R_1 \cup D^R_2 \cup \ldots \cup D^R_V$. The rule domains must be disjoint and the specification domain must be the same as the function domain. This ensures that a unique rule R_G is applicable for all data elements for which the function *F* is defined. This requirement makes the functional specification *R* unambiguous and is necessary for describing functions which are by definition unambiguous (recall the definition of a function - a function *f* from a set A to a set B is a rule which assigns to each element of A a unique element of B). The *rule computation* C^R_G is a description of the function mapping specified by the rule R_G.

If the specification is of the input/output assertion type, then the rule computation is a system of constraints defining the range of the output variables. If an operational specification is being analyzed, the rule computation is a series of statements of the function computed by the sequence of actions applied by the rule. In either case, the rule computation can be represented as a vector of N *rule functions,* $C_G^R = (r_{G1}, r_{G2}, \ldots, r_{GN})$, where the Jth component, which is either constraints or actions, provides the output of z_J.

The similarity of the characteristics associated with program paths and specification rules allows the comparison of a path and a rule for consistency. The consistency between a program and a specification may be discussed in terms of the consistency relationships between the paths in the program and the rules in the specification. With this comparison in mind, there are three consistency properties between a program and a spcification - compatibility, equivalence, and isomorphism - which are useful.

The most basic form of consistency is the compatibility of a program and a functional specification. Assuming that all logical inputs and outputs are considered - that is, there are no hidden variables - in both the program and the specification, this property must hold in order for any comparison of a program and a specification to be meaningful.

Definition - A program P is *compatible* with a specification R if P and R have the same vector of input variables $x = (x_1, x_2, \ldots, x_M)$, the same vector of output variables $z = (z_1, z_2, \ldots, z_N)$, and the same domain $X = X_1 \times X_2 \times \ldots \times X_M$, where X_I is the domain of the input variable x_I.

The compatibility property is a reasonable restriction on the class of related programs and specifications for further comparison. It requires that information represented by a single input or output variable in one entity (program or specification) be represented by a single variable in a comparable entity - this eliminates packing or mapping several variables into one variable in only one of the units for comparison.

The second property of consistency is the equivalence of a program and a functional specification. The class for which this property can be applied is the class of compatible programs and specifications. If the specification R is correct, the program P is correct if and only if this property holds.

Definition - A program P is *equivalent* to a specification R if P is compatible with R and for all $x \in X$, $P(x) = R(x)$.

This property can be stated in terms of a relationship between the paths in a program and the rules in a specification. Let $d \in X$ and suppose that $d \in D_H^P$

in the program P and $d \in D_G^R$ in the specification R - that is, d causes execution of the path P_H and the rule R_G is applicable to d. Then $P(d) = R(d)$ if and only if $C_H^P(d) = C_G^R(d)$. Since a data element d is a member of one and only one of the U path domains and one and only one of the V rule domains, the domain X may be represented as the union of the intersections of all path domains and all rule domains (this assumes that the program domain D^P and the specification domain D^R are both equal to the domain X). Let $D(P_H,R_G) = D_H^P \cap D_G^R$ denote the intersection of a path domain and a rule domain. Then, $X = \cup\, D(P_H,R_G)$ over $1 \leq H \leq U$ and $1 \leq G \leq V$. A path computation C_H^P and a rule computation C_G^R are equivalent over the intersection of their domains $D(P_H,R_G)$, if each of the path functions p_{HJ} computes the same value as the corresponding rule function r_{GJ} for each element of that domain. Thus $C_H^P \equiv C_G^R$ if and only if for all $x \in D(P_H,R_G)$, $p_{HJ}(x) = r_{GJ}(x)$, $1 \leq J \leq N$. The equivalence of a program and a specification can be redefined with these expanded definitions of domains and computations.

Definition - A program P is *equivalent* to a specification R if P is compatible with R and for all $D(P_H,R_G) = D_H^P \cap D_G^R$, $1 \leq H \leq U$ and $1 \leq G \leq V$, $C_H^P \equiv C_G^R$ over $D(P_H,R_G)$.

Equivalence is certainly the most important consistency relation between a program and a functional specification, since it holds whenever the program is correct (provided the specification is correct). Equivalence is sometimes, however, very difficult (if not impossible) to determine. Isomorphism is a restricted concept of equivalence, and as such an easier property to determine. In addition, it is often helpful to know, not only whether or not a program and a specification yield the same output for all inputs, but whether or not the program and the specification produce this value in the same manner.

This stricter form of consistency is the isomorphism of a program and a functional specification. This property is applicable to a more restricted class of programs and functional specifications. If the specification R is correct, this property is sufficient, but not necessary, for the program P to be correct.

Definition - A program P is *isomorphic* to a specification R if P is compatible with R and there exists a total bijective (one-to-one and onto) mapping I: $R \rightarrow P$ (from the rules in the specification R to the paths in the program P), such that if $I(R_K) = P_K$, then the rule R_K is "identical to" the path P_K - that is, D_K^R is "identical to" D_K^P and C_K^R is "identical to" C_K^P. The isomorphism may be defined by $I = \{(R_K,P_K)$ such that $I(R_K) = P_K\}$.

The property of isomorphism is contingent on the notion of "identical to", which will vary depending on the form of the functional specification. The identicalness of the actions applied by the specification and the sequence of statements executed by the program is legitimate only if the functional specification is an operational specification. In this case, identicalness can be interpreted as the symbolic equality of the afore-mentioned functions (which will be symbolic representations of the output variables in terms of the input variables) and domains (which will be represented as symbolic constraints on the input variables). This symbolic equality gives evidence that the output was produced in similar manners by both the program and the specification, and will be discussed more formally by proposing a method for its determination. The bijective mapping defines a one-to-one correspondence between the paths in the program and the rules in the specification. This implies that the program has the same number of paths as there are rules in the specification. A less restrictive mapping from a specification to a program might be defined in which a specification rule is mapped into a set of program paths which differ only in the number of iterations of a loop. This would allow a specification to contain a recurrence relation as a description of a loop. Cheatham (2) has suggested the use of recurrence relations in loop analysis.

The distinction between equivalence and isomorphism as consistency properties allows the attachment of differing requisites on the conformity of a program to a specification. An isomorphism might be desirable when the specification is a program design and the program is to be coded directly from the specification. In many cases, however, this strict conformity is relaxable, allowing the program merely to serve the purpose of realizing the function that the specification describes. This might occur when the specification was designed for simplicity and the program was implemented with efficiency in mind. In this case, the property of equivalence, but not isomorphism, must prevail.

Comparison of Programs and Specifications

The definitions of the equivalence and isomorphism of a program and a functional specification suggest methods for determining whether these consistency properties prevail. The compatibility of a program and a functional specification is easily determined (as such, this property will be assumed to prevail and will not be considered further). A program and a functional specification can be checked for additional consistency by comparing the characteristics of the program paths with those of the specification rules, the path domains with the rule domains and the path computations with the rule computations. In most cases, complete consistency (either equivalence or isomorphism) of a program and a functional specification is decidable only if there are a finite number of paths and rules. In general, a program (or specification) which contains indefinitely-iterated loops will contain an effectively infinite number of paths. In this case, when finiteness is not met, partial consistency of a subset of the paths and rules may be considered. (in the indeterminate loop case, this might be approached by choosing specific iterations of the loop for testing). Another approach, which was mentioned earlier, involves the specification of loops by recurrence relations which might then be compared with the corresponding loops in the program. In the discussion that follows, however, the class of programs and specifications considered is limited to those which satisfy the finiteness requirement. In addition, the class of specifications is constrained to those which provide explicitly-stated actions to be applied - that is, the class of operational specifications.

Symbolic execution of a program path can be used to generate the path condition - a system of constraints on the program's input values which describes the path domain. A set of expressions describing the path computation - symbolic representations of the program's output variables in terms of the input values - can also be constructed through this symbolic execution of the path. Since the specifications being considered are operational in form, a similar procedure can be used to construct representations of the rule domains and rule computations of the specification. Then, if the paths in a test program could be "paired up" with the rules in the operational specification, the symbolic representations of the path domains and computations could be compared with those of the rule domains and computations for either equivalence or isomorphism. First, the concept of canonical forms for the representations of the domains and computations must be considered.

A path or rule domain is represented by a set of constraints on the input variables x, which defines the subset of the function domain for which the path or rule is applicable. A path or rule domain is composed of those data elements of the domain X which satisfy all of the constraints. The constraints defining the path domain D^P_H result from the generation of the path condition $PC[P_H]$. The rule condition $RC[R_G]$ defining the rule domain D^R_G can be obtained from a similar procedure and, therefore, both the path domain D^P_H and the rule domain D^R_G may take on the same canonical form.

The canonical representation of a domain is a simplified conjunctive normal form of constraints. The expressions within the constraints are in some determined canonical form, the individual constraints are in conjunctive normal form (CNF is a conjunct of disjuncts of literals), and the individual constraints are conjoined together and ordered in some determined (but arbitrary) fashion to form one condition defining the subdomain. In addition, the conjunction may be simplified by deletion of any duplicate or redundant constraints and substitution of any equalities into other constraints in which the expression occurs. The program verifier developed by Deutsch (6) has incorporated these transformations. This simplification process may be as complex or as simple as desired, as long as the resulting canonical form is well-defined. The more extensive these transformations are, the more thorough and accurate the comparison of a path domain and a rule domain can be.

A path or rule computation is represented by a vector of formulas for the output variables. Each formula is a symbolic expression in terms of the input variables x, and represents the computation of one of the N output variables z_J. Any path computation C^P_H is represented by the vector of path functions $(p_{H1},p_{H2},\ldots,p_{HN})$, where $z_J = p_{HJ}$. Likewise, any rule computation C^R_G is represented by $(r_{G1},r_{G2},\ldots,r_{GN})$, where $z_J = r_{GJ}$. These computational formulas may be transformed into the same canonical form.

The canonical representation of a path or rule computation is a set of canonical functions. The transformation of the functions must be deterministic and might very well be the same reduction process used for the expressions in the constraints. If the functions are multivariate polynomials, this reduction might be performed by SYMPLR, a system for SYmbolic Multivariate Polynomial Linearization and Reduction (15). In the least, the transformation of the expressions into canonical form must include combination of common terms and expansion of operations to obtain simple terms.

With these canonical forms for the representations of path and rule domains and computations, comparison of the path characteristics and rule characteristics can be treated, in an attempt to determine the level of consistency which holds. The isomorphism of a program and an operational specification is easier to determine than their equivalence, so a method for determining the existence of isomorphism will be considered first.

A program is isomorphic to an operational specification if there is a one-to-one correspondence between the paths in the program and the rules in the specification such that the corresponding domains are identical and the corresponding computations are identical. Identicalness, is an easily decidable property when the representations have been transformed into a canonical form, since identicalness has been defined as symbolic equality.

When the canonical representations for the domains and computations of a program and of an operational specification are available, the question of whether the program is isomorphic to the specification is determined by attempting to "pair up" identical rules and paths. A rule and a path are identical if the representations of their domains and the representations of their computations match symbolically. A symbolic match is acheived by a term-by-term comparison of each expression for identity. If either match cannot be made, the isomorphism property does not hold. If the "pairing" is complete and the corresponding domains and computations are identical,then the program and the specification are isomorphic.

The isomorphism property might be desired when a program is to conform as closely as possible to an operational specification which has been provided. The procedure above is capable of determining when a program does conform strictly (in the manner described) to an operational specification (provided the finiteness restriction holds). In many cases, however, this strict conformity is relaxable, alowing the program to merely serve the purpose of realizing the function which the specification describes. In this case, the property of equivalence, but not isomorphism, must prevail.

A program is equivalent to an operational specification if for each non-empty subset of the function domain formed by the intersection of a path domain and a rule domain, the path computation is equivalent to the rule computation over this intersection. The decidability of this property hinges on the determination of the equivalence of computations over a specific domain, which is not always possible.

Attempting to decide whether a program and an operational specification are equivalent starts with the canonical representations of the path domains and the rule domains. The intersection of each path-rule domain pair must be constructed by conjoining the two representations. If any intersection is empty, then the path-rule pair may be discarded, since they are not mutually satisfiable - no data element exists which causes execution of the path and for which the rule is applicable. For any non-empty intersection, the corresponding path computation and rule computation must be compared for equivalence over the intersection. Two computations are equivalent over a subset of the function domain if their symbolic difference has zero value over this domain. If the path computation and the rule computation are equivalent over each non-empty path-rule domain intersection, then the program and the specification are equivalent.

One of the difficulties in this procedure is involved in deciding whether an intersection of a path domain and a rule domain is empty or non-empty, which cannot in general be determined. If all the constraints are linear, the domain may be checked for emptiness by attempting to find a solution using linear programming techniques - if the system of constraints is infeasible, then the conjunct of constraints defining the domain is unsatisfiable and the intersection is empty. In addition, deciding whether the path computation C_H^P is equivalent to the rule computation C_G^R over the intersection of the path domain and the rule domain, $D(P_H,R_G)$, is a hard problem. This decision amounts to determining whether the symbolic differences $(p_{HJ} - r_{GJ})$, $1 \leq J \leq N$ have zero value over $D(P_H,R_G)$. This decision can be made in certain situations.

A straight-forward method for trying to determine whether the symbolic difference of two functions is zero over the intersection of their domains is that of solving the equation $(p_{HJ} - r_{GJ}) = 0$ and then deciding whether the domain $D(P_H,R_G)$ is a subset of this set of solutions to the symbolic difference equation. There are several mathematical packages which contain routines for finding the zeroes of polynomials and systems of nonlinear equations, for instance (16). Determining whether the intersection domain $D(P_H,R_G)$ is contained in the solution set of the symbolic difference equation $(p_{HJ} - r_{GJ}) = 0$ may be done by showing that the intersection of these two sets is equal to $D(P_H,R_G)$.

If the constraints in the intersection domain $D(P_H,R_G)$, as well as the symbolic difference $(p_{HJ} - r_{GJ})$, are linear, then linear programming techniques can be used to show that the symbolic difference equation is zero over $D(P_H,R_G)$. This is done by determing whether $(p_{HJ} - r_{GJ}) = 0$ is valid over $D(P_H,R_G)$. Validity implies that for every data element $x \in D(P_H,R_G)$, $(p_{HJ} - r_{GJ}) = 0$. If $(p_{HJ} - r_{GJ}) \neq 0$ has no solution in $D(P_H,R_G)$ - this is the case when the condition $[((p_{HJ} - r_{GJ}) \neq 0) \wedge D(P_H,R_G)]$ is infeasible - then p_{HJ} is equivalent to r_{GJ} over the domain $D(P_H,R_G)$. If it is found, however, that $(p_{HJ} - r_{GJ}) = 0$ is not valid over $D(P_H,R_G)$, linear programming techniques can be used to determine whether or not $p_{HJ} = r_{GJ}$ is satisfiable over $D(P_H,R_G)$. Satisfiability implies that there exists a data element $d \in D(P_H,R_G)$ such that $p_{HJ} = r_{GJ}$. If the condition $[((p_{HJ} - r_{GJ}) =0) \wedge D(P_H,R_G)]$ has a feasible solution, then $p_{HJ} = r_{GJ}$ is true for some elements of $D(P_H,R_G)$. When the symbolic difference equation is completely unsatisfiable, a gross error has occurred and will be easier to detect, whereas when it is merely invalid (satisfiable), it may be more difficult to find the error (this will be discussed somewhat in a later section).

Another situation in which attempts can be made at showing that p_{HJ} and r_{GJ} are equivalent over the domain $D(P_H,R_G)$ occurs when the intersection domain is discrete. In this case, each data element $x \in D(P_H,R_G)$ can be substituted into the symbolic difference equation $(p_{HJ} - r_{GJ}) = 0$; any element for which the value of this expression is not zero will not result with equal values under the functions p_{HJ} and r_{GJ}. If for all data elements $x \in D(P_H,R_G)$, $(p_{HJ} - r_{GJ}) = 0$, then $p_{HJ} = r_{GJ}$ is valid over $D(P_H,R_G)$. Otherwise, the equality of p_{HJ} and r_{GJ} is satisfiable if there exists a $d \in D(P_H,R_G)$ such that $(p_{HJ} - r_{GJ}) = 0$; if no such data element exists, the symbolic difference equation is unsatisfiable over the intersection domain.

Steps toward determining whether the computations are equivalent over the intersection domain might also be gained by comparing the *measure* of the intersection domain $D(P_H,R_G)$ with the measure of the domain over which the symbolic difference is zero. A measure of a set is suggested by the common notions of length, area, volume, and so forth; an elementary treatment of measure theory appears in (1). If $D(P_H,R_G)$ has a larger measure than the solution set of $(p_{HJ} - r_{GJ}) = 0$, then $D(P_H,R_G)$ cannot possibly be contained in the domain over which the path and rule functions are equivalent. The pursual of research in defining measures of the domains being considered and methods for determining values of these measures is necessary.

In case none of the methods above are applicable or successful, some random test data elements out of the intersection domain may be substituted into the symbolic difference equation, checking for non-zero values. If any random data element $d \in D(P_H, R_G)$ is such that $(p_{HJ} - r_{GJ}) \neq 0$ then the functions p_{HJ} and r_{GJ} are not equivalent over the intersection domain. Evaluation of the symbolic difference equation over a limited number of random test points in $D(P_H, R_G)$ enables the attachment of a higher probability that the path and rule functions are equal over that domain. The number of test points required to assure that the functions are equivalent is dependent on the degree of the symbolic difference equation when this equation is a polynomial. For instance, if the symbolic difference is a univariate polynomial of maximal degree t which has the value zero for t + 1 unique data elements, then it is identically zero. Howden (10) has extended this property to multivariate polynomials, although for a polynomial in k variables of maximal degree t, this method requires its evaluation for $O(t^k)$ unique test data points. DeMillo and Lipton (5) have presented probabilistic results on limiting this exponential number of evaluations. Another area of research is the approximation of equivalence acheived by choosing random test points for other types of functions.

Program Errors and Inconsistency with Specifications

Program errors can be discussed in terms of the inconsistency between a program and an operational specification, just as program correctness was discussed in terms of consistency. Any discrepancy between a program and a specification may be considered from either of two perspectives - cause or effect. The effects of a program error are the inconsistencies which testing discloses, whereas the cause of an error is the actual source which debugging will hopefully discover.

If a program is inconsistent with an operational specification for the intended function of the program, then there must be at least one path which is inconsistent with the corresponding rule(s) in the specification. The effects of a program error can therefore be related to the effects on the path domains and path computations of the program. These effects are representable in terms of the inconsistencies of the path characteristics with the rule domains and computations of an operational specification. Any inconsistency of a program and an operational specification might be a non-equivalence or a non-isomorphism. Lack of equivalence implies that the program is incorrect - that it does not realize its intended function - while lack of isomorphism only implies that the strict conformity of the program to the operational specification mentioned previously does not prevail. As with determining consistency, however, much more can be decided when the presence or lack of an isomorphism is considered.

Suppose a program *P* and an operational specification *R* (both as previously defined) are expected to conform so closely as to be ismorphic. Yet, upon applying the procedure for determining whether *P* and *R* are isomorphic, a complete one-to-one correspondence between identical rules in *R* and paths in *P* cannot be found. Four simple classes of discrepancies can be defined in terms of the partial correspondence which can be obtained. These differences between the program's path characteristics and the specification's rule characteristics are called discrepancies as opposed to errors in order to emphasize the fact that they do not imply that the program is incorrect. Similar treatments of the classification of program errors have been presented by Goodenough and Gerhart (7) and Howden (8).

When a condition requires a unique sequence of computations to be processed properly, but the program fails to test for this condition, then the program is missing a path. A missing path discrepancy occurs when the function computed along a path is proper for only a subset of the domain which causes the path to be executed, and the path for which the rest of the path domain should cause execution is not present in the program.

Definition - Let P be a program intended to compute a function F on domain X and let R be a correct operational specification. Suppose there is an isomorphism I between a subset of the rules in R and all of the paths in P, such that for all $(R_K,P_K) \in I$, C^P_K is identical to C^R_K, but for some $(R_H,P_H) \in I$, D^R_H is a proper subset of D^P_H, and $D^P_H = D^R_H \cup D^R_G$ for some rule R_G which is not in I. Then P contains a *missing path discrepancy*.

On the other hand, the program may process a condition improperly due to treating it as a unique case, while the specification does not discern between this condition and another. An extraneous path discrepancy occurs when the function computed along a path is proper for the domain which causes this path to execute as well as proper for the domain which causes yet another path to execute, and the additional path is not specified in the operational specification.

Definition - Let P be a program intended to compute a function F on domain X and let R be a correct operational specification. Suppose there is an isomorphism I between all of the rules in R and a subset of the paths in P, such that for all $(R_K,P_K) \in I$, C^P_K is identical to C^R_K, but for some $(R_G,P_G) \in I$, D^P_G is a proper subset of D^R_G, and $D^R_G = D^P_G \cup D^P_H$ for some path P_H which is not in I. Then P contains an *extraneous path discrepancy*.

When a path selection predicate is expressed differently causing a series of actions to be performed (or omitted) under conditions not included in the specification, the path domain is a variant of the rule domain - that is, not identical to the rule domain. This may also occur as a result of an unspecified or missing action which affects a path selection predicate and hence the flow of control in the program. A domain discrepancy occurs when the proper function is computed along a path, but a subset of the input domain different from that indicated in the specification causes the path to be executed.

Definition - Let P be a program intended to compute a function F on domain X and let R be a correct operational specification. Suppose there is an isomorphism I between all of the rules in R and all of the paths in P, such that for all $(R_K,P_K) \in I$, C^P_K is identical to C^R_K, but for some $(R_H,P_H) \in I$, D^P_H is not identical to D^R_H, but neither domain is a subset of the other. Then P contains a *domain discrepancy.*

When an unspecified action is executed or a specified action is missing (along a path in a program) which affects the output of the program, the path computation is a variant of the rule computation - that is, not identical to the rule computation. A computation discrepancy occurs when the correct subset of the input domain causes a path to be executed, but the function computed along the path is discrepant.

Definition - Let P be a program intended to compute a function F on domain X and let R be a correct operational specification. Suppose there is an isomorphism I between all of the rules in R and all of the paths in P, such that for all $(R_K,P_K) \in I$, D^P_K is identical to D^R_K, but for some $(R_G,P_G) \in I$, C^P_G is not identical to C^R_G. Then P contains a *computation discrepancy.*

The types of discrepancies defined above can be related directly to the procedure for determining whether a program and an operational specification are isomorphic. When a program contains a computation discrepancy, a bijective mapping from the rule domains to identical path domains can be found, but one pair of corresponding rule and path computations are not identical. When a program contains a domain discrepancy, a missing path discrepancy, or an extraneous path discrepancy, a mapping from the rule computations to the path computations can be found. This mapping is one-to-one and onto in the case of a domain discrepancy, but at least one pair of corresponding rule and path domains are not identical. In the case of a missing path discrepancy, this mapping is onto the set of paths, but not defined for one of the rules, and for one pair of corresponding domains, the rule domain is a subset of the path domain. Conversely, in the case of an extraneous path discrepancy, the mapping from the rule computations to the path computations is one-to-one but not onto, and for one rule-path pair, the path domain is a subset of the rule domain. Note that in terms of the identicalness of domains under consideration, the representation of a subset of a domain will be missing at least one constraint which occurs in the conjuction which represents the larger domain.

As stated previously, the presence of a discrepancy of the type classified above does not necessarily imply that the program is incorrect. The disclosure of an inconsistency of this kind simply acknowledges the absence of strict conformity between the program and the specification. When a discrepancy is uncovered, however, in a program which was intended to conform exactly to an operational specification, the rule-path pair for which the discrepancy occurs should be examined more closely. At this point, the discrepancy could be corrected, or perhaps a check for the equivalence of the variant characteristics could be performed.

When a program is not designed for strict conformity with an operational specification which describes the intended function, it is not legitimate to draw conclusions about errors (or even discrepancies) from a subjection of the two entities to the procedure to determine whether isomorphism holds. It may, however, be helpful to apply this procedure and note all rule-path correspondences which satisfy identicality. An isomorphic rule-path pair need not be considered in the application of the equivalence determination procedure, because isomorphic rule-path pairs are certainly equivalent. In addition, the rule domain and path domain of any isomorphic rule-path pair need not be intersected with other path and rule domains when applying the equivalence procedure, since the correspondence implies that the rule domain is identical to the path domain and all path domains are disjoint and all rule domains are disjoint (due to the determinism of programs and specifications). More formally, if the path P_K is isomorphic to the rule R_K, then D^P_K is identical to D^R_K, so $D^P_K = D^R_K$, but for all $H \neq K$, $1 \leq H \leq U$, $D^P_K \cap D^P_H = \phi$, and for all $G \neq K$, $1 \leq G \leq V$, $D^R_K \cap D^R_G = \phi$, therefore for all $H \neq K$, $D^P_H \cap D^R_K = \phi$ and for all $G \neq K$, $D^P_K \cap D^R_G = \phi$. Deciding upon the prevalence of isomorphism is much easier than the desision of whether or not equivalence holds, hence any gains which can be made through application of the isomorphism procedure are extremely beneficial. After noting any identical pairs of rules and paths, the procedure for determining the equivalence of the program and the specification must be applied to check for errors among the remaining paths and rules.

In considering a lack of equivalence between a program and an operational specification, the errors which may occur have not been uniquely defined in terms of their effects on the path domains and computations. In general, an error occurs on any path for which the path computation is not identical to the computation of a rule whose corresponding domain is not disjoint from the path domain. The same division into four errors can be made in terms of the inconsistencies between a program and an operational specification which are not equivalent. Precise definitions for these in terms of the differences between rule and path characteristics should be researched in more depth.

The classification of inconsistencies presented does not encompass all possible variant programs. The variant programs considered contain only single inconsistencies rather than combinations of inconsistencies. In addition, variants caused by incorrect declarartions are not considered, although declarations in a program might also be compared with any similar construct in the functional specification in order to detect such differences. A discrepancy between the variables (declared or not) in a program and a specification would imply a lack of compatibility between the two entities, and thus neither equivalence nor isomorphism could hold.

Error Detection With Symbolic Execution and Test Data Generation Systems

The effects of errors can also be considered as the actual discrepancies between the outputs of the program and the intended values of the corresponding function outputs. With actual execution, an error is detected when an output value is incorrect. Unaided actual execution has long been the only practical method of attesting to the correctness of a program. Automatic test data generation is a valuable aid to the human tester by replacing the floundering selection of test data. Test data generation is most often accomplished by a symbolic execution system which also provides another facility for error detection.

Symbolic execution systems, such as ATTEST (3), symbolically execute the paths in a program to create symbolic representations of the path domains and path computations. A program may be tested by examining these representations. Whenever a domain-type error or a computation error occurs, the symbolic representation of the path domain, or path computation respectively, will be incorrect. Hence, symbolic execution systems will, in some sense, uncover all errors in a program of the four classes defined. It is often difficult, however, for a system user to determine when a symbolic representation is incorrect. The comparison techniques discussed may provide some advances in the detection of errors by this approach.

In the absence of automated comparison of the path characteristics with rule characteristics of a specification, test data must be relied upon to disclose errors. By finding a solution to the symbolic representations of a path domain, test data is generated which will cause execution of the path. In most current automatic test data generation systems, an approach called path testing is used, in which a single data element is provided for the testing of a path. A set of test data whose elements will cause different paths in the program to be tested will be generated. Ideally, this test data set includes a data element for each path in the program; a problem arises, however, when the number of paths is infinite or effectively so, and heuristics must be used to select a subset of the paths for testing. For example, an infinite number of possible paths in a program occurs when a loop exists which has an indeterminate loop iteration count. Automatic path selection heuristics might choose paths which execute this path the minimum number of times, the maximum number of times, and a few intermediate number of iterations. The techniques of path testing cannot reliably detect all errors, but does provide a high degree of assurance that the program has been well-tested (8).

A computation error will always be detected if the error occurs for all elements of the path domain. If the error occurs for only a subset of the path domain, however, coincidental correctness may result by exercising the path on an element for which the path computation happens to be correct.

A domain error on a path will always be detected if the path domain in the test program is disjoint from the correct path domain. If the two domains intersect, then coincidental correctness may result by exercising the path on an element of the intersection, causing the correct path to be executed and no error to occur.

A missing path error may not be detected (no restrictions on the domain will insure its detection). This is due to the fact that test data generation systems may choose an element of the path domain which is in the correct domain and should not be a member of the missing path's domain. The error will be detected if an element of the missing path's correct domain happens to be selected for a test data element.

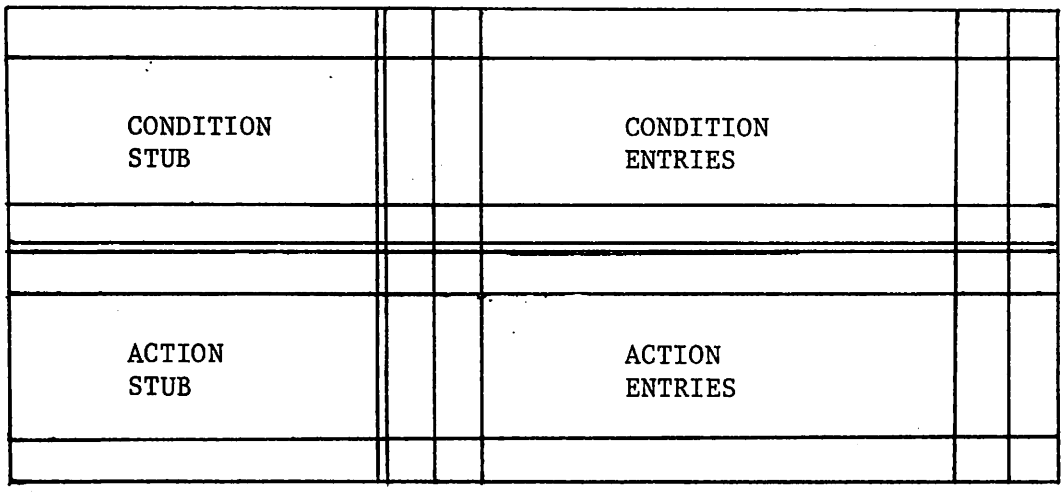
An extraneous path error will always be detected, provided the path is in the selected subset of paths tested and the path computation is distinct from the computation which should be executed. Extraneous implies that the path should not be executed for any element of the input domain. Hence, exercising the path on any element of the path domain will reveal the error.

Test data generation systems are capable of selecting more than one test data point per execution path, and this approach might enable a more reliable detection of these classes of errors. For example, checking the boundaries of path domains might enable a better chance of revealing domain-type errors. Cohen and White (4) have developed a domain-testing strategy based on this idea. As mentioned in comparing computations for equivalence over a domain, the choice of a certain number of random test points might enable a greater assurance that a computation is correct over a path domain.

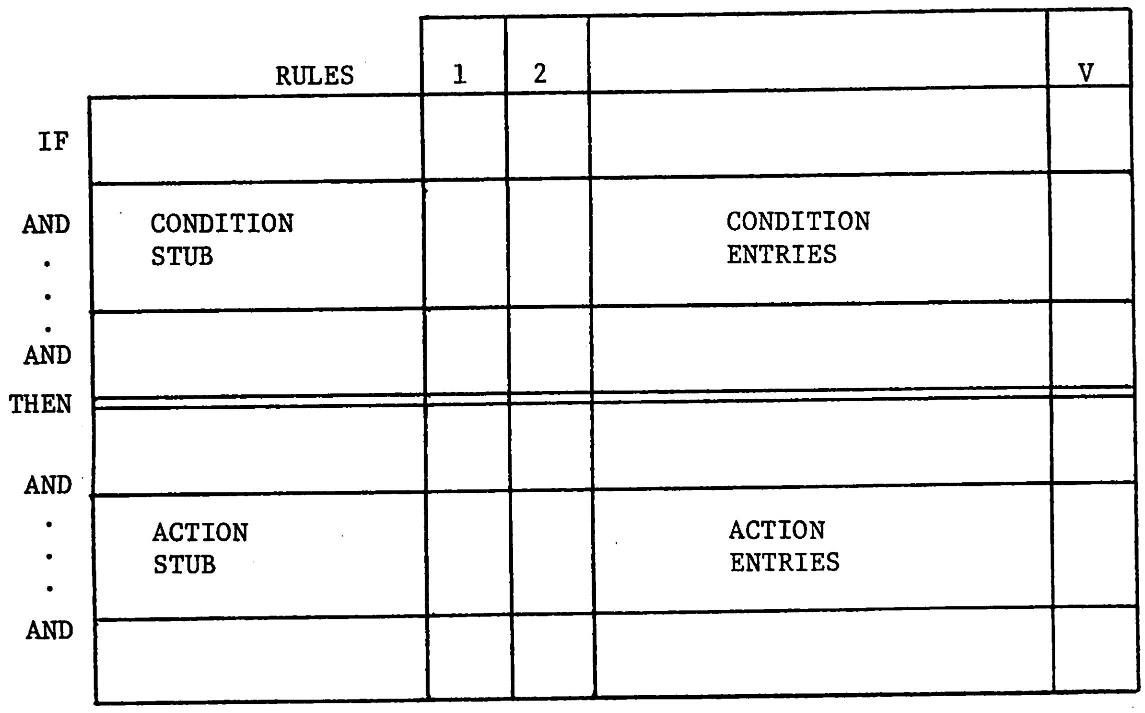
Possibilities of Decision Tables in Program Testing

A decision table provides a convenient form for expressing any conditional alternatives, where a particular computation to be performed is dictated by the outcome of a combination of conditions. A decision table (13) is a tabular representation of an operational specification and the comparison techniques discussed previously can be readily applied to the decision table structure. In addition, another method for using decision tables in program testing is proposed below, which can be generalized to a technique for using other forms of operational specifications.

The basic structure of a decision table is shown below.



A decision table is divided into four quadrants. The condition stub in the upper left quadrant contains all those conditions which have bearing on the decisions in the particular problem. The upper right quadrant contains the condition entries which specify the relevant outcomes of the combination of conditions. The action stub in the lower left specifies explicitly the actions that may be applied in obtaining output for the problem, while the action entries in the lower right specify the actions to be applied under certain conditions. A rule is a column in the decision table, which represents the relationship between an outcome of the combination of conditions and a set of actions. The meaning of these different sections is shown below as an if-then construct.



Two types of decision tables will be considered. The major difference between the two is the variables on which the decisions can be made.

The first type of decision table contains decisions based on the input variables only. All conditions on the input variables which are considered relevant to the problem are written in the condition stub. Each column of the table represents a specific interpretation of these predicates - a particular combination of conditions. In the action stub of the decision table, the transformations to be applied are explicitly stated as either symbolic computations or individual actions which should be performed in an ordered sequence. The action entries specify the actions which should be performed for the corresponding combination of conditions specified in the condition entry of this rule.

The second type of decision table contains decisions based on both the input variables and the local variables. All conditions on the program variables (both input and local) considered relevant to the problem are written in the condition stub, and each column represents an interpretation of these predicates. The action stub of this type of decision may represent the actions in either of the ways mentioned previously.

The two forms of specifying the actions are equivalent. Indicating a sequence of actions to be performed provides a better program specification - more of a design that could be translated into a program. A sequence can however be transformed into computations similar to the other form by a technique like symbolic execution, and this symbolic computation appears to be easier to handle in program testing.

As for the different methods of specifying conditions and decisions - with input variables only or including local variables as well - they too are essentially equivalent. As with the two types of action representations, one method - conditions on both the input and local variables - provides a program design approximation. Conditions on the input domain only provide specifications for the input domain of a specific rule. Both methods can be more useful in particular situations. Conditions on the input domain alone can be obtained from conditions on both input and local variables by the symbolic execution technique.

There are several ways in which decision tables can enhance the process of testing a program. The simplest method (which applies only to decision tables containing predicates on the input variables only) is obtaining test data for each rule by solving the combination of conditions which determine the domain for which the rule applies. For a particular rule, the appropriate computation (determined by the action entry) can be applied to the test data obtained. The program itself can be run on the test data, and the output obtained from both can then be compared for equality. Any incorrect values obtained by the program should be noted and the programmer should take a closer look at the path executed.

Other methods include the application of the isomorphism and equivalence procedures presented earlier. The procedure for determining whether the program and the decision table are isomorphic can be applied when the decision table was intended as a design to which the program should conform. If a one-to-one correspondence can be created between identical rules in the decision table and program paths, then the isomorphism property prevails. This can be determined by defining the rule domains by the conjunction of conditions specified as true in the condition entry and the rule computation as that obtained by applying the actions indicated in the action entry. If there is a bijective mapping from each rule in the decision table to a path in the program for which the rule domain is identical to the path domain and the rule computation is identical to the path computation, then the two are isomorphic and the program is correct.

If isomorphism does not or was not intended to hold, the procedure for determining whether the program is equivalent to the decision table may be attempted. For a particular path in the program, some combinations of conditions in the decision table ought to be consistent with the path domain. If no composite predicate is consistent, then an error is present in the program (assuming the decision table is a correct specification for the intended function). For each of the combinations of conditions which is consistent with the path domain, determine the intersection of the two domains. At this point, the path computation can be compared to the computation in the rule determined by the combination of conditions. If these are found to be equivalent over the intersection domain, no errors have been detected. If this computation equivalence holds for all of the composite predicates which have non-empty intersections with the path domain, no errors have been detected along this path. If the computations are not equivalent over one of the intersection domains, an error has been disclosed. Performing this for each path in the program will allow the determination of the equivalence between the test program and the decision table specification.

A further method begins with the rules in the decision table specification. Given a particular rule, the combination of conditions can be used to select a path in the program during symbolic execution of the program. At each branch point in the program, each interpretation of the path selection predicate will be tested for consistency with the combined conditions of the rule (plus any other conditions which have been added). If one interpretation is consistent and all others are not, then the decision at this point is determined and symbolic execution continues along the path in the direction of the branch selected. If none of the interpretations are consistent (this can only happen with an incomplete computed go to or case statement), then an error is present in the program (again, assuming the decision table is a correct operational specification). At this point, the comparison may stop and the tester informed of the error, or symbolic execution may be continued along all branches in order to analyze the error in some way. Suppose, on the other hand, that more than one interpretation is consistent with the combination of conditions, then symbolic execution should be executed along all consistent branches. Along each consistent branch (when there is more than one), the current combination of conditions must be supplemented by the interpretation of the branch predicate, this will provide the intersection of the decision table rule conditions and the path domain which causes execution of the path. After an entire path has been selected - that is, all branch decisions have been made down to an exit statement - the path has been symbolically executed, and symbolic expressions for the output variables in terms of the input variables have been obtained. These path computations can then be compared with the action rules in the decision table. If these are equivalent over the domain determined by the combination of conditions (the rule conditions plus any indeterminate branch decisions chosen), no errors have been detected along the path. If the computations are not equivalent, an error is present and a closer look at the path (both computations and predicates) should be suggested. If the lack of errors can be shown for each rule in the decision table with each consistent path selection, the program is equivalent to the decision table specification.

There are problems inherent in this procedure. In addition to the undecidable properties of computation equivalence, there is a problem with expanding the symbolic execution along all consistent branches when a decision is not determined by the conditions - namely, indeterminate loop iterations. If all consistent branches are taken (in the case of loop iterations, this implies another iteration and leaving the loop), the number of paths under symbolic execution for a single decision table rule (with indeterminacy) will quickly explode. This problem could be approached by applying the heuristics of automatic path selection, as mentioned above, and comparing the path computation of these paths with the actions specified in the decision table rule.

Decision tables provide a good example of the form of an operational specification, and the methods for using decision tables can be generalized to techniques for using other forms of operational specifications in program testing.

An Example of Equivalence and Isomorphism

The specification below is rather abstract, yet it is a complete operational specification for a 3 × 3 matrix multiplication routine.

$$\text{FOR } i=1,3 \ (\text{FOR } j=1,3 \ (C_{ij} \leftarrow \sum_{k=1}^{3} A_{ik} * B_{kj}))$$

If there were a symbolic interpreter for this type of specification, the rule computations which would result would be those shown below, which are already in simplest terms.

```
C(1,1) = A(1,1)*B(1,1) + A(1,2)*B(2,1) + A(1,3)*B(3,1)
C(1,2) = A(1,1)*B(1,2) + A(1,2)*B(2,2) + A(1,3)*B(3,2)
C(1,3) = A(1,1)*B(1,3) + A(1,2)*B(2,3) + A(1,3)*B(3,3)
C(2,1) = A(2,1)*B(1,1) + A(2,2)*B(2,1) + A(2,3)*B(3,1)
C(2,2) = A(2,1)*B(1,2) + A(2,2)*B(2,2) + A(2,3)*B(3,2)
C(2,3) = A(2,1)*B(1,3) + A(2,2)*B(2,3) + A(2,3)*B(3,3)
C(3,1) = A(3,1)*B(1,1) + A(3,2)*B(2,1) + A(3,3)*B(3,1)
C(3,2) = A(3,1)*B(1,2) + A(3,2)*B(2,2) + A(3,3)*B(3,2)
C(3,3) = A(3,1)*B(1,3) + A(3,2)*B(2,3) + A(3,3)*B(3,3)
```

The FORTRAN subroutine shown below implements this specification in the most straight-forward manner. This subroutine is both equivalent and isomorphic to the specification given, as one would expect.

```
      SUBROUTINE MM3X3(A,B,C)
      DIMENSION A(3,3),B(3,3),C(3,3)
      DO 10 I = 1,3
         DO 10 J = 1,3
            C(I,J) = 0.0
            DO 10 K = 1,3
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
   10          CONTINUE
      RETURN
      END
```

A symbolic execution system like ATTEST (3) provides path functions which are exactly like those produced from symbolic interpretation of the specification. The symbolic difference between the corresponding path and rule functions is identically zero, and hence these two entities are equivalent. In addition, since they match symbolically, the isomorphism property holds as well.

The next FORTRAN subroutine is a correct implementation of a 3 × 3 matrix multiplication, but uses Laderman's algorithm which performs only 23 multiplications (as opposed to the 27 performed in the usual algorithm).

```
SUBROUTINE LADER(A,B,C)
DIMENSION A(3,3),B(3,3),C(3,3)
REAL M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12,M13
REAL M14,M15,M16,M17,M18,M19,M20,M21,M22,M23
M1=(A(1,1)+A(1,2)+A(1,3)-A(2,1)-A(2,2)-A(3,2)-A(3,3))*B(2,2)
M2=(A(1,1)-A(2,1))*(-B(1,2)+B(2,2))
M3=A(2,2)*(-B(1,1)+B(1,2)+B(2,1)-B(2,2)-B(2,3)-B(3,1)+B(3,3))
M4=(-A(1,1)+A(2,1)+A(2,2))*(B(1,1)-B(1,2)+B(2,2))
M5=(A(2,1)+A(2,2))*(-B(1,1)+B(1,2))
M6=A(1,1)*B(1,1)
M7=(-A(1,1)+A(3,1)+A(3,2))*(B(1,1)-B(1,3)+B(2,3))
M8=(-A(1,1)+A(3,1))*(B(1,3)-B(2,3))
M9=(A(3,1)+A(3,2))*(-B(1,1)+B(1,3))
M10=(A(1,1)+A(1,2)+A(1,3)-A(2,2)-A(2,3)-A(3,1)-A(3,2))*B(2,3)
M11=A(3,2)*(-B(1,1)+B(1,3)+B(2,1)-B(2,2)-B(2,3)-B(3,1)+B(3,2))
M12=(-A(1,3)+A(3,2)+A(3,3))*(B(2,2)+B(3,1)-B(3,2))
M13=(A(1,3)-A(3,3))*(B(2,2)-B(3,2))
M14=A(1,3)*B(3,1)
M15=(A(3,2)+A(3,3))*(-B(3,1)+B(3,2))
M16=(-A(1,3)+A(2,2)+A(2,3))*(B(2,3)+B(3,1)-B(3,3))
M17=(A(1,3)-A(2,3))*(B(2,3)-B(3,3))
M18=(A(2,2)+A(2,3))*(-B(3,1)+B(3,3))
M19=A(1,2)*B(2,1)
M20=A(2,3)*B(3,2)
M21=A(2,1)*B(1,3)
M22=A(3,1)*B(1,2)
M23=A(3,3)*B(3,3)
C(1,1)=M6+M14+M19
C(1,2)=M1+M4+M5+M6+M12+M14+M15
C(1,3)=M6+M7+M9+M10+M14+M16+M18
C(2,1)=M2+M3+M4+M6+M14+M16+M17
C(2,2)=M2+M4+M5+M6+M20
C(2,3)=M14+M16+M17+M18+M21
C(3,1)=M6+M7+M8+M11+M12+M13+M14
C(3,2)=M12+M13+M14+M15+M22
C(3,3)=M6+M7+M8+M9+M23
RETURN
END
```

Symbolic execution of this subroutine provides the path computation - vector of path functions - which appears on the following page. Using a polynomial simplification system such as SYMPLR (15), these symbolic representations reduce (convert to a canonical form) to the same functions produced by the operational specification for a 3 × 3 matrix multiplication. Thus, after the transformation to canonical form, the path functions of LADER are equivalent as well as symbolically identical to those of the subroutine MM3X3 and the rule functions of the specification. The program LADER is, therefore, both equivalent and isomorphic to the operational specification given. The fact that there is only one path through the subroutine has made the determination quite easy. In addition, this is the reason that a program which is so seemingly different from the specification is still isomorphic to it; for if isomorphism did not hold, the program would not be equivalent to the specification, and thus not a correct program. Since the domain over which the two computations must be equal is the entire domain, any discrepancy between the two symbolic computations would correspond to an error.

```
C(1,1) = (A(1,1)*B(1,1) + (A(1,3)*B(3,1)) + (A(1,2)*B(2,1))
C(1,2) = (A(1,1)*B(2,2)+A(1,2)*B(2,2)+A(1,3)*B(2,2)-A(2,1)*B(2,2)
         -A(2,2)*B(2,2)-A(3,2)*B(2,2)) + (-A(1,1)*B(1,1)+A(1,1)*B(1,2)
         -A(1,1)*B(2,2)+A(2,1)*B(1,1)-A(2,1)*B(1,2)+A(2,1)*B(2,2)
         +A(2,2)*B(1,1)-A(2,2)*B(1,2)+A(2,2)*B(2,2) + (-A(2,1)*B(1,1)
         +A(2,1)*B(1,2)-A(2,2)*B(1,1)+A(2,2)*B(1,2)) + (A(1,1)*B(1,1))
         + (-A(1,3)*B(2,2) - A(1,3)*B(3,1)+A(1,3)*B(3,2)+A(3,2)*B(2,2)
         +A(3,2)*B(3,1)-A(3,2)*B(3,2)+A(3,3)*B(2,2)+A(3,3)*B(3,1)
         -A(3,3)*B(3,2)) + (A(1,3)*B(3,1)) + (-A(3,2)*B(3,1)
         +A(3,2)*B(3,2)-A(3,3)*B(3,1)+A(3,3)*B(3,2)
C(1,3) = (A(1,1)*B(1,1)) + (-A(1,1)*B(1,1)+A(1,1)*B(1,3)-A(1,1)*B(2,3)
         +A(3,1)*B(1,1)-A(3,1)*B(1,3)+A(3,1)*B(2,3)+A(3,2)*B(1,1)
         -A(3,2)*B(1,3)+A(3,2)*B(2,3) + (-A(3,1)*B(1,1)+A(3,1)*B(1,3)
         -A(3,2)*B(1,1)+A(3,2)*B(1,3) + (A(1,1)*B(2,3)+A(1,2)*B(2,3)
         +A(1,3)*B(2,3)-A(2,2)*B(2,3)-A(2,3)*B(2,3)-A(3,1)*B(2,3)
         -A(3,2)*B(2,3)) + (A(1,3)*B(3,1)) + (-A(1,3)*B(2,3)-A(1,3)*B(3,1)
         +A(1,3)*B(3,3)+A(2,2)*B(2,3)+A(2,2)*B(3,1)-A(2,2)*B(3,3)
         +A(2,3)*B(2,3)+A(2,3)*B(3,1)-A(2,3)*B(3,3)) + (-A(2,2)*B(3,1)
         +A(2,2)*B(3,3)-A(2,3)*B(3,1)+A(2,3)*B(3,3))
C(2,1) = (-A(1,1)*B(1,2)+A(1,1)*B(2,2)+A(2,1)*B(1,2)-A(1,2)*B(2,2))
         + (-A(2,2)*B(1,1)+A(2,2)*B(1,2)+A(2,2)*B(2,1)-A(2,2)*B(2,2)
         -A(2,2)*B(2,3)-A(2,2)*B(3,1)+A(2,2)*B(3,3)) + (-A(1,1)*B(1,1)
         +A(1,1)*B(1,2)-A(1,1)*B(2,2)+A(2,1)*B(1,1)-A(2,1)*B(1,2)
         +A(2,1)*B(2,2)+A(2,2)*B(1,1)-A(2,2)*B(1,2)+A(2,2)*B(2,2))
         + (A(1,1)*B(1,1)) + (A(1,3)*B(3,1)) + (-A(1,3)*B(2,3)-A(1,3)*B(3,1)
         +A(1,3)*B(3,3)+A(2,2)*B(2,3)+A(2,2)*B(3,1)-A(2,2)*B(3,3)
         +A(2,3)*B(3,2)+A(2,3)*B(3,1)-A(2,3)*B(3,3)) + (A(1,3)*B(2,3)
         -A(1,3)*B(3,3)-A(2,3)*B(2,3)+A(2,3)*B(3,3))
C(2,2) = (-A(1,1)*B(1,2)+A(1,1)*B(2,2)+A(2,1)*B(1,2)-A(2,1)*B(2,2))
         + (-A(1,1)*B(1,1)+A(1,1)*B(1,2)-A(1,1)*B(2,2)+A(2,1)*B(1,1)
         -A(2,1)*B(1,2)+A(2,1)*B(2,2)+A(2,2)*B(1,1)-A(2,2)*B(1,2)
         +A(2,2)*B(2,2)) + (-A(2,1)*B(1,1)+A(2,1)*B(1,2)-A(2,2)*B(1,1)
         +A(2,2)*B(1,2)) + (A(1,1)*B(1,1)) + (A(2,3)*B(3,2))
C(2,3) = (A(1,3)*B(3,1)) + (-A(1,3)*B(2,3)-A(1,3)*B(3,1)+A(1,3)*B(3,3)
         +A(2,2)*B(2,3)+A(2,2)*B(3,1)-A(2,2)*B(3,3)+A(2,3)*B(2,3)
         +A(2,3)*B(3,1)-A(2,3)*B(3,3)) + (A(1,3)*B(2,3)-A(1,3)*B(3,3)
         -A(2,3)*B(2,3)+A(2,3)*B(3,3)) + (-A(2,2)*B(3,1)+A(2,2)*B(3,3)
         -A(2,3)*B(3,1)+A(2,3)*B(3,3)) + (A(2,1)*B(1,3))
```

```
C(3,1) = (A(1,1)*B(1,1)) + (-A(1,1)*B(1,1)+A(1,1)*B(1,3)-A(1,1)*B(2,3)
         +A(3,1)*B(1,1)-A(3,1)*B(1,3)+A(3,1)*B(2,3)+A(3,2)*B(1,1)
         -A(3,2)*B(1,3)+A(3,2)*B(2,3)) + (-A(1,1)*B(1,3)+A(1,1)*B(2,3)
         +A(3,1)*B(1,3)-A(3,1)*B(2,3)) + (-a(3,2)*B(1,1)+A(3,2)*B(1,3)
         +A(3,2)*B(2,1)-A(3,2)*B(2,2)-A(3,2)*B(2,3)-A(3,2)*B(3,1)
         +A(3,2)*B(3,2)) + (-A(1,3)*B(2,2)-A(1,3)*B(3,1)+A(1,3)*B(3,2)
         +A(3,2)*B(2,2)+A(3,2)*B(3,1)-A(3,2)*B(3,2)+A(3,3)*B(2,2)
         +A(3,3)*B(3,1)-A(3,3)*B(3,2)) + (A(1,3)*B(2,2)-A(1,3)*B(3,2)
         -A(3,3)*B(2,2)+A(3,3)*B(3,2)) + (A(1,3)*B(3,1))

C(3,2) = (-A(1,3)*B(2,2)-A(1,3)*B(3,1)+A(1,3)*B(3,2)+A(3,2)*B(2,2)
         +A(3,2)*B(3,1)-A(3,2)*B(3,2)+A(3,3)*B(2,2)+A(3,3)*B(3,1)
         -A(3,3)*B(3,2)) + (A(1,3)*B(2,2)-A(1,3)*B(3,2)-A(3,3)*B(2,2)
         +A(3,3)*B(3,2)) + (A(1,3)*B(3,1)) + (-A(3,2)*B(3,1)+A(3,2)*B(3,2)
         -A(3,3)*B(3,1)+A(3,3)*B(3,2)) + (A(3,1)*B(1,2))

C(3,3) = (A(1,1)*B(1,1)) + (-A(1,1)*B(1,1)+A(1,1)*B(1,3)-A(1,1)*B(2,3)
         +A(3,1)*B(1,1)-A(3,1)*B(1,3)+A(3,1)*B(2,3)+A(3,2)*B(1,1)
         -A(3,2)*B(1,3)+A(3,2)*B(2,3)) + (-A(1,1)*B(1,3)+A(1,1)*B(2,3)
         +A(3,1)*B(1,3)-A(3,1)*B(2,3)) + (-A(3,1)*B(1,1)+A(3,1)*B(1,3)
         -A(3,2)*B(1,1)+A(3,2)*B(1,3)) + (A(3,3)*B(3,3))
```

Future Research

Several areas have been mentioned in which research in the field of program testing is necessary. With regard to program correctness and consistency with functional specifications, several open questions were presented in the consistency properties and the procedures for determining whether they prevail. How "identical" are programs designed to conform to an operational specification? Can comparisons be made between loops in a program and recurrence relations in a functional specification? When can the determination of emptiness of a domain be decided? When can the equivalence of computations over a domain be determined? With what types of computations can approximations of the determinism of equivalence be acheived by choosing a certain number of random test points? The applicability of both measure theory and approximation algorithms should be pursued. In addition, it may be helpful to place further restrictions on the classes of programs and speicications analyzed. For instance, the complexity of determining equivalence of Ianov and free schema programs to specifications and approximation algorithms for this decision should be examined. With regard to program errors, a more extensive classification of program errors is required, including definitions of errors in programs which are not equivalent to functional specifications (without regarding the lack of isomorphism). Empirical studies of common programs, operational specifications, programming errors, and the implication of the procedures presented for determining isomorphism and equivalence must be conducted.

References

1. Bartle, Robert G., The Elements of Integration, John Wiley and Sons, Inc., New York, 1966.

2. Cheatham, Thomas E., Jr., and Deborah A. Washington, "Program Loop Analysis by Solving First Order Recurrence Relations", SIAM-SIGSAM Computer Algebra Symposium, May, 1978.

3. Clarke, Lori A., "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, September, 1976.

4. Cohen, Edward I., and Lee J. White, "A Finite Domain-Testing Strategy for Computer Program Testing", Ohio State University CISRC Technical Report # 77-13, August, 1977.

5. DeMillo, Richard A., and Richard J. Lipton, "A Probabilistic Remark on Algebraic Program Testing", School of Information and Computer Science Technical Report, Georgia Institute of Technology, May, 1977.

6. Deutsch, L. Peter, "An Interactive Program Verifier", Ph. D. Dissertation, University of California, Berkeley, May, 1973.

7. Floyd, Robert W., "Assigning Meaning to Programs", Proceedings of the American Mathematical Society Symposium on Applied Mathematics, Vol. 19, 1967.

8. Goodenough, John B., and Susan L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, June, 1975.

9. Howden, William E., "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, September, 1976.

10. Howden, William E., "Algebraic Program Testing", Acta Informatica, to appear.

11. Liskov, Barbara H., and Valdis Berzins, "An Appraisal of Program Specifications", MIT Technical Report, April, 1977.

12. Manna, Zohar, Mathematical Theory of Computation, McGraw-Hill, Inc., New York, 1974.

13. Pooch, Udo W., "Translation of Decision Tables", ACM Computing Surveys, June, 1974.

14. Richardson, Debra J., "Some Aspects of Program Testing", Masters Thesis, University of Massachusetts, Amherst, May, 1978.

15. Richardson, Debra J., Lori A. Clarke, and Debi L. Bennett, "SYMPLR, SYmbolic Multivariate Polynomial Linearization and Reduction", University of Massachusetts COINS Technical Report #78-16, July, 1978.

16. University of Wisconsin Academic Computing Center, "Mathematical Routines Directory".